HIVE: A MULTI-AGENT INFRASTRUCTURE FOR ALGORITHM- AND TASK-LEVEL SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly deployed as complex agentic systems. While prior work has focused on model- and system-level scaling, algorithm- and task-level scaling remain largely unaddressed. At the algorithm level, test-time scaling enhances reasoning but introduces **cross-path redundancy**: branches expanded from the same state re-generate overlapping tokens, yet existing engines execute them as independent requests. At the task level, agents exhibit highly heterogeneous runtime characteristics, yet current inference engines schedule all requests uniformly without awareness of agent roles. We propose *Hive*, a multi-agent inference infrastructure that enables algorithm- and task-level scaling. *Hive* provides a description front-end that captures per-agent behavior and supports test-time scaling algorithms. Its back-end introduces LOGITS CACHE for algorithm-level scaling, which reuses intermediate logits across redundant sampling paths to mitigate cross-path redundancy, and AGENT-AWARE RESOURCE ALLOCATION for task-level scaling, which prioritizes KV-cache retention according to agent contributions. Experiments show that LOGITS CACHE achieves an average speedup of $1.11\times$–$2.51\times$, and AGENT-AWARE RESOURCE ALLOCATION reduces the hotspot miss rate by 33%–49%.

1 INTRODUCTION

Large language models (LLMs) have advanced rapidly in recent years (Team et al., 2026; Qwen Team, 2026; Liu et al., 2025; OpenAI, 2025b), demonstrating strong capabilities in reasoning, generation, and decision-making. As their competence continues to grow, their role is also shifting from single-turn response engines to agentic problem solvers that can plan over multiple steps, invoke external tools, and coordinate toward complex goals (OpenClaw, 2026; Anthropic, 2025; OpenAI, 2025a). In these emerging workloads, inference is no longer a collection of independent prompt-response pairs. Instead, it becomes a stateful process in which context accumulates across turns, intermediate results are repeatedly reused, and multiple reasoning branches or agents may coexist within a single task.

As problem complexity continues to grow, effectively solving real-world tasks often requires scaling LLM systems along multiple dimensions rather than relying on a fixed model and static inference procedure. To better characterize this trend, we summarize the scaling landscape of LLM systems into four complementary paradigms:

- **Model-Level Scaling** improves the capability of LLMs by increasing model size and fine-tuning with domain-specific datasets, typically leading to stronger reasoning and generation ability.
- **System-Level Scaling** improves serving efficiency through additional hardware resources and parallelism, enabling higher throughput, lower latency, and better scalability.
- **Algorithm-Level Scaling** improves the potential of the same workflow by allocating additional computation at inference time, e.g., test-time scaling (Zhang et al., 2025b).
- **Task-Level Scaling** improves the scalability and robustness of problem solving by decomposing complex objectives into subproblems, often coordinated through multi-agent systems.

However, optimization in existing inference systems has been concentrated mainly on model-level and system-level scaling. At the model level, substantial effort has been devoted to quantization and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

compression (Zhang et al., 2025a; Chen et al., 2024; Huang et al., 2025; Chen et al., 2025), improving model-hardware efficiency. At the system level, serving architectures and deployment strategies such as PD-disaggregation (Zhong et al., 2024; Patel et al., 2024) and context parallelism (Liu et al., 2023; Jacobs et al., 2023) have been developed to improve hardware utilization. By contrast, little attention has been paid to runtime support for algorithm-level and task-level scaling, even though both are becoming increasingly important in agentic workloads.

At the algorithm level, the inherent stochasticity of large language models is often unacceptable in high-assurance domains, making test-time scaling essential for reliability. Test-time scaling methods such as Tree-of-Thoughts (Yao et al., 2023) organize reasoning as a branching search over intermediate states. The LLM expands a state into candidate branches, evaluates them, and continues exploring the accepted ones. When a branch is rejected, the search backtracks to the parent state and re-expands another branch from its full context. Yet, this approach introduces **Cross-Path Redundancy**. Branches expanded from the same state share the entire prefix context, and even beyond the shared prefix they tend to re-generate overlapping tokens. This is because along the shared prefix the logits distributions are exactly identical and high-probability tokens remain likely to be sampled again. Prefix caching (Zheng et al., 2024; Gao et al., 2024) eliminates only the repeated computation over the shared context. Each branch still requires a full forward pass at every decode step to produce its logits again, since existing inference engines (Zheng et al., 2024; Kwon et al., 2023) treat the branches as independent requests.

Task-level scaling further enhances agentic systems by delegating subtasks to specialized agents, each handling a well-defined role and coordinating through multi-turn interactions. Such role specialization naturally produces different workload characteristics among agents (Li et al., 2024; Tran et al., 2025). This raises a new infrastructure challenge. Agents exhibit **heterogeneous runtime characteristics** in input/output lengths, cache footprints, and context reuse patterns. For instance, an orchestrator agent typically consumes far more tokens than the specialized subagents it supervises, while a context-gathering agent repeatedly queries a long shared context. Multi-agent execution also introduces substantially higher request concurrency and more complex inter-agent dependencies than conventional single-request inference. This places much stronger pressure on KV-cache management and serving throughput. Current inference engines remain largely agent-unaware. Under a generic eviction policy such as LRU, eviction is determined solely by access recency. KV-cache states of frequently reused agents may therefore be evicted simply because they are less recent than newly inserted states, even though they are far more likely to be reused in subsequent execution. Such agent-agnostic eviction can significantly degrade overall efficiency in multi-agent workloads.

In this paper, we present *Hive*, a multi-agent inference infrastructure that rethinks how modern inference engines support algorithm-level and task-level scaling. The key insight behind *Hive* is that the inefficiency of scaling workloads does not merely arise from increased compute demand, but also from the fact that existing inference engines are not designed to fully exploit their structural regularities. *Hive* is composed of a multi-agent description front-end and LLM inference back-end. The front-end captures the per-agent behavior, supports test-time-scaling algorithms, and models multi-agent interaction as asynchronous coroutine-like task spawning and synchronization. In the back-end, for algorithm-level scaling, *Hive* introduces LOGITS CACHE, a new caching paradigm that reuses the logits of intermediate states across redundant sampling paths, avoiding repeated resampling computation. For task-level scaling, *Hive* introduces the AGENT-AWARE RESOURCE ALLOCATION mechanism, which prioritizes KV-cache retention according to agent-specific runtime characteristics, improving cache hit rates and overall serving utilization under multi-agent workloads.

Experiments show that LOGITS CACHE achieves an average speedup of $1.11\times$–$2.51\times$ for resampling on the Hugging Face 24 Game dataset (Nlile, 2026), and AGENT-AWARE RESOURCE ALLOCATION reduces the hotspot miss rate by 33%–49% on the R^3A (Luo et al., 2025) multi-agent workflow, compared with the original mechanisms of SGLang (Zheng et al., 2024).

The main contributions of this paper are as follows:

- We propose *Hive*, a multi-agent inference infrastructure that supports algorithm- and task-level scaling by modeling multi-agent and test-time scaling in the front-end and jointly optimizing inference execution in the back-end.

- For **algorithm-level scaling**, we propose a new caching paradigm called LOGITS CACHE for hybrid test-time scaling to reduce redundant computation.

- For **task-level scaling**, we identify distinctive heterogeneity in multi-agent workloads, and propose agent-aware resource allocation that prioritizes KV-cache residency according to agent-specific characteristics.

2 BACKGROUND AND RELATED WORK

Modern LLMs (Team et al., 2026; Qwen Team, 2026; Liu et al., 2025) are typically served through autoregressive inference with two stages: a *prefill* stage that processes the prompt and constructs the KV cache, and a *decode* stage that generates tokens one by one, sampling each next token from the logits distribution produced at that step. Decode reuses the cached keys and values from previous steps, avoiding recomputation over the full prefix. This split execution model makes KV cache management, scheduling, and resource allocation central to high-performance LLM inference systems (Zhong et al., 2024; Agrawal et al., 2024; Patel et al., 2024). More aggressive decoding strategies such as speculative inference (Miao et al., 2024) can further improve efficiency at the system level.

Beyond serving individual requests efficiently, a second direction scales the computation spent at the algorithm level: as training-time scaling (Hoffmann et al., 2022; Kaplan et al., 2020) slows down due to the high cost of pretraining and limited high-quality data, *test-time scaling* (Zhang et al., 2025b) instead allocates additional computation at inference time. It generally takes three forms: *parallel* (Li et al., 2025; Wang et al., 2025) scaling samples multiple responses to the same prompt, *sequential* (Madaan et al., 2023; Bi et al., 2024) scaling iteratively refines previous responses, and *hybrid* (Yao et al., 2022; 2023) scaling combines both into a tree. Recent work has started to improve algorithm-level scaling from a systems perspective. For example, FastTTS (Chen et al., 2026) studies execution scheduling and memory management for verifier-guided test-time scaling, aiming to improve serving efficiency under more complex inference workloads. DEFT (Yao et al., 2025) proposes a KV-guided grouping strategy to improve I/O reuse in tree-based decoding. *Hive* primarily focuses on eliminating redundant resampling in *hybrid* test-time scaling in this work, which is orthogonal to these related works.

Furthermore, task-level scaling enables multi-role orchestration for more complex tasks. A multi-agent system (MAS) consists of multiple interacting agents collaborating toward a common objective, widely regarded as a pathway to collective intelligence while preserving per-agent specialization (Li et al., 2024; Tran et al., 2025; Hu et al., 2021; PolyArch, 2026). KVFlow (Pan et al., 2025) and TokenCake (Bian et al., 2025) study KV-cache management for *multi-agent* LLM serving. KVFlow proposes a workflow-aware mechanism that constructs an *Agent Step Graph* offline from the workflow structure, and assigns each agent a graph-derived timestamp to guide KV eviction and prefetching, thereby improving prefix reuse in structured agentic workloads. TokenCake (Bian et al., 2025) presents a KV-cache-centric serving framework for multi-agent applications, providing a DSL to describe inter-agent dependencies as a DAG and using this structure to drive a *Time Scheduler* and a *Space Scheduler* for execution coordination and KV-cache allocation across agents. ThunderAgent (Kang et al., 2026) mainly focuses on *single-agent* execution. It introduces a program-aware scheduling framework that jointly manages KV cache, program state, and tool resources. In contrast, *Hive* focuses on runtime contribution-aware resource allocation for dynamic multi-agent serving, allocating KV-cache resources according to the observed runtime behavior and relative contribution of different agents.

3 OVERVIEW

To address these challenges, we introduce *Hive*, an inference framework that efficiently supports algorithm- and task-level scaling workloads, as illustrated in Figure 1. *Hive* is organized into a descriptive front-end that expresses agents and test-time scaling algorithms, and an execution backend that jointly optimizes the two scaling paradigms at runtime.

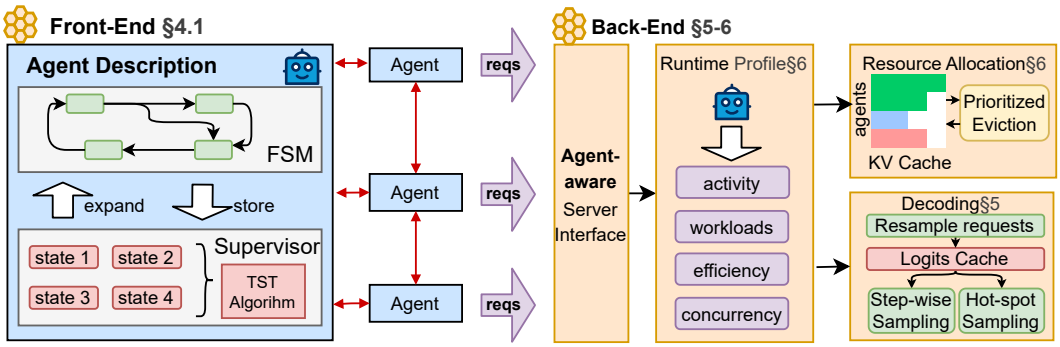


Figure 1: Overview of Hive Multi-Agent Infrastructure

3.1 Front End

Hive describes a multi-agent system at two levels: a **finite state machine** (FSM) for each agent and a **supervisor** for each test-time scaling (TTS) algorithm.

Each agent's behavior is captured by an FSM. A **state** represents a continuation of the agent. During one execution step, the agent transits to another state, inheriting the data and dialogue states for the LLMs. Furthermore, each agent is equipped with a **supervisor**, which implements the agent's TTS algorithm by deciding which FSM states to execute and responding to their results. This separation keeps agent behavior orthogonal to TTS algorithms, enhancing code reuse. The front-end also annotates each request with metadata, such as the identity of the issuing agent and the continuation of the tree-based context. A complete description of the front-end abstraction and its asynchronous runtime is provided in Appendix C.

3.2 Back End

The back-end serves the requests emitted by the front-end as a single inference pipeline, as illustrated in Figure 1. The server interface accepts each request and binds it to the agent that issued it, which makes the runtime behavior of every agent observable at the request level for task-level scaling. This binding enables **Runtime Profiling**, which characterizes each agent by its activity, workload, efficiency, and concurrency.

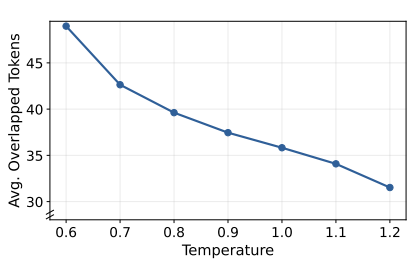
The resulting profiles drive Agent-Aware Resource Allocation (Section 5), which sets KV-cache residency according to the contribution of the owning agent. The tree-based context metadata enables Logits Cache (Section 4), which serves resample requests from the logits of previously visited states and thereby reduces cross-path redundancy at the algorithm level.

4 Logits Cache

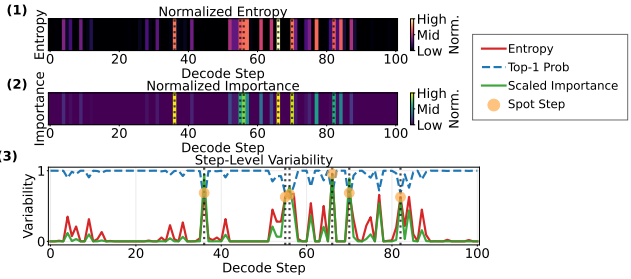
To reduce cross-path redundancy in hybrid test-time scaling, we introduce Logits Cache, which makes the logits at an intermediate state reusable across repeated expansions. Repeated expansions are executed independently yet often traverse highly similar decode trajectories and produce overlapping tokens. The next-step logits at a revisited state are thus both expensive to regenerate and sufficiently informative to be reused.

Formally, at decode step t the transformer-based model (Vaswani et al., 2017) maps the current decoding state to a next-token logits vector via Eq 1, where x_t is the current input token, $\mathrm{KV}_{<t}$ the cached key-value states of the prefix, h_t the hidden representation, and z_t the logits over the vocabulary. Although the KV cache (Gao et al., 2024) avoids recomputing the shared prefix, producing z_t still requires a full decode computation. Eliminating this redundant computation frees bandwidth and compute capacity for concurrent requests.

Hive supports two resampling policies, *Step-wise Sampling* and *Hotspot Sampling*, described next, followed by the overall Logits Cache workflow.



(a) Average token overlap with step-wise sampling under different temperatures.



(b) Step-level logit uncertainty on a 100-token decoding trace generated at temperature 0.6.

Figure 2: Profiles for logit cache on R3A (Luo et al., 2025). For token overlap, *top-p*=1 and *top-k* unconstrained, excluding additional sampling truncation effects. For logit uncertainty: (1) Normalized entropy per decode step. (2) Normalized hotspot importance score. (3) Step-level variability statistics and the selected hotspot steps.

4.1 Step-wise Sampling

Upon a Logits Cache hit, *Hive* traverses the cached logits sequence and samples along the cached trajectory under the sampling configuration, appending each sampled token to the request until the sampled token diverges from the cached continuation. On divergence, the replayed tokens are concatenated with the original input prefix and normal prefill and decode resume.

Step-wise Sampling preserves the original autoregressive execution order and introduces no approximation beyond logits reuse, retaining the full information of the cached sequence. In practice, however, strict token-by-token replay recovers only a limited reusable prefix (Figure 2a), which constrains the achievable reduction in branch-expansion latency.

4.2 Hotspot Sampling

Hotspot Sampling instead resamples only the subset of positions most likely to alter the downstream decode trajectory. This design follows an empirical observation: uncertainty is concentrated in a small fraction of decode steps, with only a few positions exhibiting pronounced entropy peaks (Figure 2b(1)). These steps are thus responsible for most downstream divergence.

To identify these hotspot positions, *Hive* computes an importance score for each decode step t from its cached logits. Let $\mathbf{p}_t = \mathrm{softmax}(\mathbf{z}_t)$ be the token distribution at step t. We compute its entropy (Eq 2) and top-1 confidence (Eq 3), and define the hotspot score as in Eq 4, where λ is a time-decay factor. This score favors positions that are both uncertain and weakly peaked, while biasing selection toward earlier positions whose perturbation affects a longer downstream suffix. Since the raw score inherits the unbounded scale of the entropy term, *Hive* min-max normalizes the scores over the trace, and selects positions whose normalized score exceeds a threshold as hotspots, resampling only there and preserving the original sampled tokens at stable positions (Figure 2b(2)(3)). Hotspot Sampling thereby recovers a longer reusable prefix and further reduces post-hit decode computation. To assess the potential loss of exploration from token reuse, we evaluate answer correctness in Section 6 and examine token-level output variation across hotspot thresholds in Appendix B.

4.3 Algorithm

Hive serves a request with Logits Cache as follows. Given a request r, sampling configuration ϕ, and replay policy π, *Hive* derives the current intermediate state s and initializes a buffer $\mathcal{Z}_{new}$ for the logits generated in this execution. On a cache hit, the cached logits sequence and continuation are loaded. Under *Step-wise Sampling*, the engine samples token by token along the cached sequence until divergence. Under *Hotspot Sampling*, it identifies the hotspot positions, resamples only these, and preserves cached tokens elsewhere, also stopping early if a sampled token diverges from the cached continuation. If the request remains unfinished after replay, *Hive* prefills the current request, including any replayed tokens, to materialize its KV states and obtain the next-step logits.

If replay has completed the request, it returns directly. Otherwise normal decoding continues until completion, with each generated logit appended to the buffer, and the newly generated logits are finally written back to LOGITS CACHE for future revisits. Repeated expansions from the same state are thus reduced to a cache lookup followed by replay and resampling. Both replay policies are detailed in Appendix A.

The LOGITS CACHE is stored in host memory rather than on the GPU. The only GPU-side cost of LOGITS CACHE is writing the newly generated logits back to host memory, which incurs minor transfer overhead. Replay sampling itself runs entirely on CPU and adds no GPU computation. For efficiency, prefetch sampling runs asynchronously on CPU: once the cache is updated for the current request, the runtime immediately triggers a background prefetch task for the same state with its associated sampling configuration, overlapping cache update with replay preparation for future revisits and thereby shortening the blocking time.

$$h_t = f_\theta(x_t, \mathrm{KV}_{<t}), \qquad z_t = W_o h_t, \quad (1)$$

$$H_t = -\sum_v p_t(v) \log p_t(v) \quad (2)$$

$$p_t^{\max} = \max_v p_t(v) \quad (3)$$

$$s_t = H_t(1 - p_t^{\max}) \frac{1}{1 + \lambda t} \quad (4)$$

5 AGENT-AWARE RESOURCE ALLOCATION

To address agent-agnostic resource management in multi-agent workloads, we design AGENT-AWARE RESOURCE ALLOCATION, a run-time resource allocation policy that governs KV cache residency according to per-agent contribution. Unlike traditional engines that treat requests as uniform, *Hive* elevates resource management to the agent level: requests originate from agents with distinct functional roles and heterogeneous runtime behaviors, which contribute unevenly to the workflow. Its effect is confined to KV-cache retention and eviction under memory pressure. It does not reorder request dispatch or admission. We first introduce our contribution modeling, whose collaborative component is grounded in the Shapley value, and then describe how the policy uses these scores for dynamic KV cache eviction.

5.1 AGENT CONTRIBUTION MODELING

Existing systems ignore that agents with heterogeneous roles, execution patterns, and cache-sharing potential impact overall efficiency disproportionately. To bridge this gap, *Hive* models each agent's contribution by integrating two complementary signals. Each active agent $a \in \mathcal{A}$ receives an **Intrinsic Utility** ($\widehat{U}_a$), the standalone importance of the agent's own task progress, and a **Collaborative Utility** ($\widehat{C}_a$), the shared-cache reuse the agent facilitates for other agents, which we derive from the Shapley value of a cooperative interaction game (Rozemberczki et al., 2022). The final score is defined as in Eq 5. We empirically set $\alpha = 0.4$, prioritizing collaborative reuse for the throughput gains of shared KV cache states.

Intrinsic Utility. The intrinsic utility captures the standalone runtime importance of an agent as the product of four normalized factors (Eq 6). The activity (A_a) and workload (W_a) terms capture *inference momentum* through invocation frequency and aggregate token volume. The efficiency term (E_a) captures *resource productivity* through KV cache reuse. The concurrency term (Q_a) captures parallel demand through average and peak concurrency.

Collaborative Utility. The collaborative utility captures the benefit an agent induces for the rest of the workflow through shared-cache reuse. *Hive* measures it from the bidirectional shared-cache interaction graph induced at runtime: for each parent-child agent pair (a, b), the edge weight is defined as in Eq 8, where $\mathrm{S}_{a,b}$ is the total shared-token count and $\mathrm{Event}_{a,b}$ the number of reuse events. The square-root term prevents large token volumes from dominating, while the logarithmic factor rewards repeated collaboration.

$$\mathrm{Score}(a) = \alpha \cdot \widehat{U}_a + (1 - \alpha) \cdot \widehat{C}_a \quad (5)$$

$$U_a = A_a \cdot W_a \cdot E_a \cdot Q_a \quad (6)$$

$$C_a = \sum_{b \neq a} w(a, b) + \sum_{b \neq a} w(b, a) \quad (7)$$

$$w(a, b) = \sqrt{\mathrm{S}_{a,b}} \cdot \log(1 + \mathrm{Event}_{a,b}) \quad (8)$$

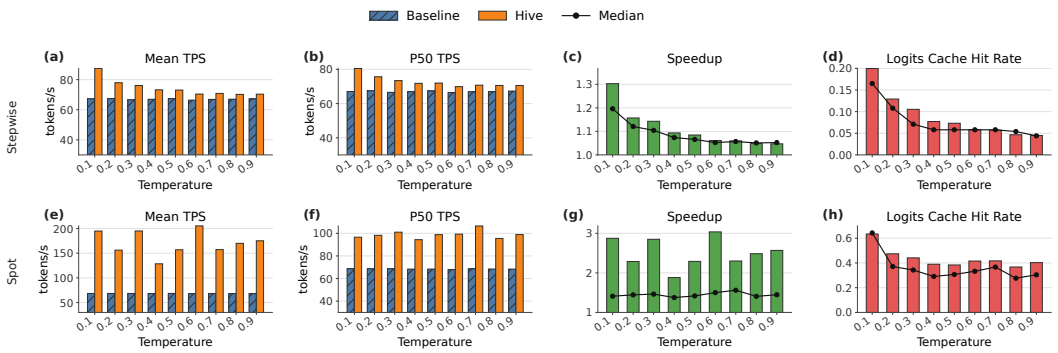


Figure 3: Performance of LOGITS CACHE under different temperatures for two replay policies. (a)–(d) show *step-wise sampling* and (e)–(h) show *hotspot sampling*.

5.2 AGENT-AWARE KV CACHE ALLOCATION

At runtime, *Hive* prioritizes KV cache by the contribution score of the owning agent: cache of higher-contribution agents is retained on the GPU whenever possible, and eviction under memory pressure is biased toward lower-contribution agents. Scores are recomputed each round from runtime profiling, with a sliding window preventing abrupt changes in agent behavior. In this way, *Hive* aligns KV cache residency with the structural importance of agents in the workflow, preserving the states that matter most for efficient multi-agent execution.

6 EVALUATION

We evaluate *Hive* from two perspectives, corresponding to the two scaling dimensions that it targets. For *algorithm-level scaling*, we evaluate LOGITS CACHE on Tree-of-Thoughts workloads to measure how effectively it reduces replay overhead during repeated resampling. For *task-level scaling*, we evaluate AGENT-AWARE RESOURCE ALLOCATION on multi-agent workloads to study whether agent-aware KV-cache allocation improves execution efficiency over conventional eviction policies. Together, these experiments demonstrate how *Hive* provides unified runtime support for both replay-efficient reasoning and scalable multi-agent execution.

6.1 EXPERIMENTAL SETUP

Models and Platforms. We evaluate the two components using separate model deployments. The LOGITS CACHE experiments use DeepSeek-V3.2 (Liu et al., 2025), deployed across two machines with 8×NVIDIA H100 GPUs each. The AGENT-AWARE RESOURCE ALLOCATION experiments use Qwen3-8B (Yang et al., 2025) on one NVIDIA A100 GPU. Its server is equipped with 8×NVIDIA A100 GPUs and an Intel Xeon Gold 6348 CPU @ 2.60 GHz. The software stack on the Qwen3-8B server includes CUDA Toolkit 11.5, PyTorch 2.9.1, Python 3.12.13, and SGLang 0.5.6.

6.2 LOGITS CACHE EVALUATION

Workload. We evaluate LOGITS CACHE on repeated resampling workloads induced by hybrid test-time scaling using the Tree-of-Thoughts dataset (Yao et al., 2023). Specifically, the initial request performs standard autoregressive decoding and records the logits produced at each step. Subsequent requests that revisit the same intermediate state resample from the cached sequence, avoiding repeated model execution. The workload is constructed from the Hugging Face 24 Game dataset (Nlile, 2026), which contains 1.36k instances of four-number arithmetic puzzles. We sample 100 instances as the evaluation workload, whose average prompt length is 64.52 tokens.

Baseline. Our baseline disables LOGITS CACHE and recomputes the full logits sequence for every replayed request. To ensure a fair comparison, the baseline and *Hive* use the same KV cache configuration and retain identical prefix-state reuse behavior. The only difference is whether the replayed request can reuse cached logits or must regenerate them through normal decoding.

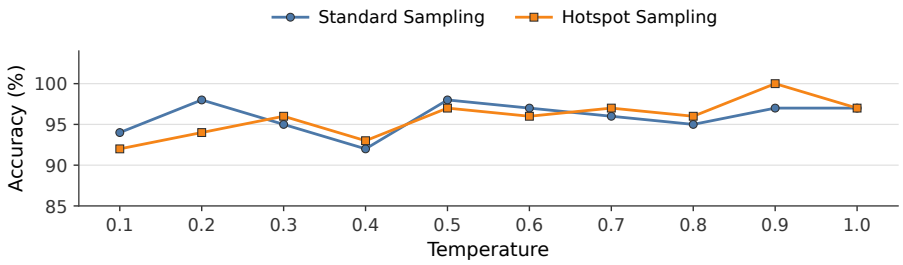


Figure 4: Final-answer accuracy of Standard Sampling and Hotspot Sampling across different temperatures.

Decoding Configuration. Unless otherwise specified, the baseline and *Hive* use identical decoding configuration, including temperature, top-p, top-k, and `max_tokens`, for a fair comparison. We consider two replay settings. Since sampling is inherently stochastic, repeated runs may produce outputs of different lengths, making the results harder to reproduce and compare fairly. To reduce this variation, we fix the output length for every request in both settings. Following (Wu et al., 2026), which reports an average output length of 429 tokens for agentic workloads, we set the fixed output length to 500 tokens. For *hotspot sampling*, decoding positions whose min-max-normalized importance score exceeds 0.6 are treated as *spot nodes*. We vary the temperature to study how the benefit of LOGITS CACHE changes across different replay conditions.

Metrics. For all reported metrics, we exclude requests with a 100% Logits Cache hit ratio, since such requests would distort the measurement results. In total, we filter out 71 requests. We then report the following metrics for LOGITS CACHE during resampling requests. **MEAN TPS** is the average tokens per second. **P50 TPS** is the median number of tokens per second. **Speedup** is the average per-request speedup. **Logits-cache hit ratio** is the fraction of decoding positions in a replayed request whose logits are served from the cache instead of being recomputed by the model.

Performance. Figure 3 presents the replay TPS of LOGITS CACHE under step-wise and hotspot replay across temperatures; it consistently exceeds the baseline, with the largest gains at low temperatures. The benefit should grow with prompt length, since a longer prefill raises the per-step computation cost that sampling from cached logits bypasses. This matters most in agentic workloads with far longer context windows. DualPath (Wu et al., 2026) reports 32.7K-token average agentic inputs, where logits-based reuse should deliver even greater gains.

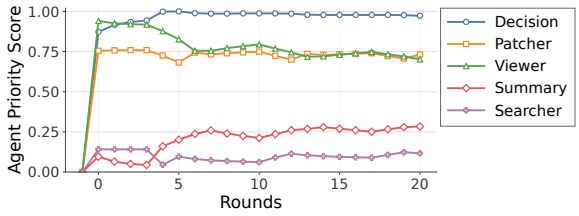
Under *step-wise sampling*, mean and P50 TPS improve most at low temperatures. As temperature rises, replayed branches diverge earlier from the cached continuation, reducing reusable logits and weakening the gain. Under *hotspot sampling*, the improvement is more pronounced, yielding much larger low-temperature speedups: operating only on high-importance positions, hotspot replay accepts larger token segments once reuse succeeds.

Hotspot replay's token acceptance also explains Figure 3(g): one reused hotspot node can trigger a cascading sequence of accepted tokens, so the average speedup exhibits higher variance. While the mean speedup fluctuates noticeably, the median remains stable and matches the expected trend.

In conclusion, LOGITS CACHE achieves an average speedup of $1.11\times$ under step-wise replay and $2.51\times$ under hotspot replay, with average Logits Cache hit rates of 8.8% and 43.6%, respectively.

Correctness across Temperatures. Beyond throughput, we verify that logits reuse preserves answer accuracy, evaluating Qwen3.8-27B on 100 fixed, solvable 24 Game instances (Nlile, 2026) across temperatures $T \in \{0.1, 0.2, \ldots, 1.0\}$, with a shared prompt, tokenizer, and model backend, top-$p = 1$, unrestricted top-k, min-$p = 0$, and normalized hotspot threshold 0.6. For each puzzle, Standard Sampling first generates an initial sequence. Under matched position-specific seeds, we then compare two new outputs: one fresh Standard sample, and one Hotspot sample reusing the initial tokens. Only these new outputs enter the correctness metric.

An output is correct if its returned arithmetic expression uses each of the four input numbers exactly once and evaluates to 24 using the permitted operations. For each method and temperature, accuracy is the proportion of correct outputs across the 100 puzzles.



(a) Normalized agent-priority score trajectories.

Round	Method	Hotspot hit rate	Non-hotspot hit rate	Evicted tokens
7	BASELINE	0.903	0.311	26557
7	HIVE	0.935	0.310	18549
21	BASELINE	0.935	0.414	81147
21	HIVE	0.967	0.311	65598

(b) KV-cache hit rates and token eviction.

Figure 5: Agent-aware resource allocation evaluation. (a) Agent priority scores over 21 rounds. (b) KV-cache hit rates and evicted tokens for the baseline and *Hive*.

Figure 4 shows that Hotspot Sampling maintains answer accuracy comparable to Standard Sampling across the evaluated temperatures. Their mean accuracies are 95.8% and 95.9%, respectively. At six of the ten temperatures, Hotspot Sampling matches or exceeds Standard Sampling, and temperature-specific gains and losses reveal no consistent pattern of degradation. These results suggest that selective token reuse has little effect on overall answer correctness on this benchmark.

6.3 AGENT-AWARE RESOURCE ALLOCATION EVALUATION

Workload. To evaluate AGENT-AWARE RESOURCE ALLOCATION under practical conditions, we construct a benchmark multi-agent workflow following the R^{3}A (Luo et al., 2025) execution pattern. This workload integrates five heterogeneous agent roles (DECISION, PATCHER, VIEWER, SUMMARY, and SEARCHER), creating highly uneven runtime demands and providing a rigorous testbed for evaluating *Hive*'s ability to balance competing agent utilities and optimize KV cache residency. To ensure reproducibility, we fix the agent invocation relationships as a static workflow during evaluation, so that all compared policies are tested under the same execution structure.

Memory Setting. We set `mem-fraction-static` to 0.23 to emulate a memory-constrained serving environment, thereby amplifying cache contention and making the impact of different eviction policies more pronounced. This value is chosen based on the model size and the prompt length used in the benchmark.

Baseline. We compare *Hive*'s agent-aware policy against the original LRU-based cache eviction policy in SGLang.

Performance. As shown in Figure 5(b), AGENT-AWARE RESOURCE ALLOCATION consistently improves KV cache efficiency over the baseline: at round 7, the hotspot hit rate rises from 0.903 to 0.935 with a $1.43\times$ reduction in evicted tokens, and by round 21 from 0.935 to 0.967 with a $1.24\times$ reduction. *Hive* thereby better preserves KV states on critical execution paths under memory pressure. Notably, the improvement comes mainly from prioritizing hotspot tokens rather than uniformly raising all hit rates, consistent with favoring high-contribution agents during cache eviction.

The per-agent score traces in Figure 5(a) further confirm this: scores evolve with each agent's runtime invocation pattern rather than remaining static or uniform. Our contribution formulation thus models the relative contribution of heterogeneous agents online, allowing the policy to adjust allocation priorities accordingly.

Overall, across different rounds, *Hive* reduces the hotspot miss rate by 33%–49% over the baseline, while reducing evicted KV tokens by 19.2%–30.2%, demonstrating the effectiveness of agent-aware resource allocation under memory pressure.

7 CONCLUSION

Current LLM infrastructures mainly scale at the model level and system level. We propose *Hive*, a multi-agent infrastructure for algorithm- and task-level scaling. Experiments show that LOGITS CACHE achieves an average speedup of $1.11\times$–$2.51\times$ for re-sampling, and AGENT-AWARE RESOURCE ALLOCATION reduces the hotspot miss rate by 33%–49%.

AI USE STATEMENT

In this work, we used generative AI tools for drafting and polishing portions of this manuscript, for identifying related work, and for writing minor parts of the code, such as visualization scripts. We have not used generative AI tools for generating synthetic datasets, helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, implementing methods, assisting with translation, cleaning and reformatting datasets, supporting qualitative and thematic data analysis, or interpreting results. The remaining required disclosure tasks are not applicable to this work. We have reviewed and verified all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve, 2024. URL https://arxiv.org/abs/2403.02310.

Anthropic. Claude Code: Agentic coding in your terminal. https://www.anthropic.com/news/claude-code, 2025. Accessed: 2026-07.

Zhenni Bi, Kai Han, Chuanjian Liu, Yehui Tang, and Yunhe Wang. Forest-of-thought: Scaling test-time compute for enhancing llm reasoning. *arXiv preprint arXiv:2412.09078*, 2024.

Zhuohang Bian, Feiyang Wu, Teng Ma, and Youwei Zhuo. Tokencake: A kv-cache-centric serving framework for llm-based multi-agent applications. *arXiv preprint arXiv:2510.18586*, 2025.

Hao Mark Chen, Zhiwen Mo, Guanxi Lu, Shuang Liang, Lingxiao Ma, Wayne Luk, and Hongxiang Fan. Fasttts: Accelerating test-time scaling for edge llm reasoning. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pp. 732–748, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723599. doi: 10.1145/3779212.3790161. URL https://doi.org/10.1145/3779212.3790161.

Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, and Ping Luo. EfficientQAT: Efficient quantization-aware training for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10081–10100, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.498. URL https://aclanthology.org/2025.acl-long.498/.

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative llm inference with recallable key-value eviction. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. doi: 10.52202/079017-3595.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention, 2024. URL https://arxiv.org/abs/2403.19708.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and L. Sifre. Training compute-optimal large language models. *ArXiv*, abs/2203.15556, 2022. URL https://api.semanticscholar.org/CorpusID:247778764.

Junyan Hu, Parijat Bhowmick, Inmo Jang, Farshad Arvin, and Alexander Lanzon. A decentralized cluster formation containment framework for multirobot systems. *IEEE Transactions on Robotics*, 37(6):1936–1955, 2021. doi: 10.1109/TRO.2021.3071615.

Wei Huang, Yi Ge, Shuai Yang, Yicheng Xiao, Huizi Mao, Yujun Lin, Hanrong Ye, Sifei Liu, Ka Chun Cheung, Hongxu Yin, et al. Qerl: Beyond efficiency–quantization-enhanced reinforcement learning for llms. *arXiv preprint arXiv:2510.11696*, 2025.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

Hao Kang, Ziyang Li, Xinyu Yang, Weili Xu, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. Thunderagent: A simple, fast and program-aware agentic inference system. *arXiv preprint arXiv:2602.13692*, 2026.

Jared Kaplan, Sam McCandlish, Thomas Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeff Wu, and Dario Amodei. Scaling laws for neural language models. *ArXiv*, abs/2001.08361, 2020. URL https://api.semanticscholar.org/CorpusID:210861095.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Chengpeng Li, Mingfeng Xue, Zhenru Zhang, Jiaxi Yang, Beichen Zhang, Bowen Yu, Binyuan Hui, Junyang Lin, Xiang Wang, and Dayiheng Liu. Start: Self-taught reasoner with tools. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 13523–13564, 2025.

Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9, 2024. doi: 10.1007/s44336-024-00009-2. URL https://doi.org/10.1007/s44336-024-00009-2.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Zizhang Luo, Fan Cui, Kexing Zhou, Runlin Guo, Mile Xia, Hongyuan Hou, and Yun Liang. R3a: Reliable rtl repair framework with multi-agent fault localization and stochastic tree-of-thoughts patch generation, 2025. URL https://arxiv.org/abs/2511.20090.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, volume 3 of *ASPLOS '24*, pp. 932–949. Association for Computing Machinery, 2024. ISBN 979-8-4007-0386-7. doi: 10.1145/3620666.3651335. URL https://dl.acm.org/doi/10.1145/3620666.3651335.

Nlile. 24-game, 2026. URL https://huggingface.co/datasets/nlile/24-game. Accessed: 2026-04-11.

OpenAI. Codex CLI: Open-source agentic coding in the terminal. https://github.com/openai/codex, 2025a. Accessed: 2026-07.

OpenAI. Introducing gpt-5.4. https://openai.com/index/introducing-gpt-5-4/, December 2025b. Accessed: 2026-04-15.

OpenClaw. openclaw/openclaw: Your own personal AI assistant. Any OS. Any Platform. The lobster way. https://github.com/openclaw/openclaw, 2026. GitHub repository, accessed 2026-03-24.

Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. *arXiv preprint arXiv:2507.07400*, 2025.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

PolyArch. Humanize: From automated idea factory to realization. https://github.com/PolyArch/humanize, 2026. GitHub repository, version 1.16.0, accessed August 3, 2026.

Qwen Team. Qwen3-coder-next technical report. Technical report, 2026. URL https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf. Accessed: 2026-02-03.

Benedek Rozemberczki, Lauren Watson, Péter Bayer, Hao-Tsung Yang, Olivér Kiss, Sebastian Nilsson, and Rik Sarkar. The shapley value in machine learning, 2022. URL https://arxiv.org/abs/2202.05594.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. Multi-agent collaboration mechanisms: A survey of llms, 2025. URL https://arxiv.org/abs/2501.06322.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Junlin Wang, Jue WANG, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=h0ZfDIrj7T.

Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilin Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, Mingxing Zhang, Xin Jin, and Panpan Huang. Dualpath: Breaking the storage bandwidth bottleneck in agentic llm inference, 2026. URL https://arxiv.org/abs/2602.21548.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. Deft: Decoding with flash tree-attention for efficient tree-structured llm inference. 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Jintao Zhang, Jia Wei, Pengle Zhang, Xiaoming Xu, Haofeng Huang, Haoxu Wang, Kai Jiang, Jianfei Chen, and Jun Zhu. Sageattention3: Microscaling fp4 attention for inference and an exploration of 8-bit training. *arXiv preprint arXiv:2505.11594*, 2025a.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, et al. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*, 2025b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024.

APPENDIX

A LOGITS CACHE RESAMPLING WORKFLOW

Algorithm 1: Logits Cache Resampling with Step-wise and Hotspot Sampling

Input: Input request r, Sampling configuration ϕ, Replay policy π

$s \leftarrow \text{GETSTATE}(r)$;
$\mathcal{Z}_{new} \leftarrow \emptyset$;
if $\text{HASCACHEDLOGITS}(s)$ **then**
  $\mathcal{Z}_{cache} \leftarrow \text{GETCACHEDLOGITS}(s)$;
  $\mathcal{Y}_{cache} \leftarrow \text{GETCACHEDTOKENS}(s)$;
  if $\pi = \text{STEPWISESAMPLING}$ **then**
    foreach *z in $\mathcal{Z}_{cache}$* **do**
      $y \leftarrow \text{SAMPLE}(z, \phi)$;
      $r \leftarrow \text{APPENDTOKEN}(r, y)$;
      if *y diverges from cached continuation* **then**
        break;
  else if $\pi = \text{HOTSPOTSAMPLING}$ **then**
    $\mathcal{H} \leftarrow \text{IDENTIFYHOTSPOTS}(\mathcal{Z}_{cache})$;
    foreach *position i in $\mathcal{Z}_{cache}$* **do**
      if $i \in \mathcal{H}$ **then**
        $y \leftarrow \text{SAMPLE}(\mathcal{Z}_{cache}[i], \phi)$;
      else
        $y \leftarrow \mathcal{Y}_{cache}[i]$;
      $r \leftarrow \text{APPENDTOKEN}(r, y)$;
      if *y diverges from cached continuation* **then**
        break;
if $\text{NOTFINISHED}(r)$ **then**
  $z \leftarrow \text{PREFILL}(r)$;
else
  return r;
while $\text{NOTFINISHED}(r)$ **do**
  append z to $\mathcal{Z}_{new}$;
  $y \leftarrow \text{SAMPLE}(z, \phi)$;
  $r \leftarrow \text{APPENDTOKEN}(r, y)$;
  if $\text{NOTFINISHED}(r)$ **then**
    $z \leftarrow \text{DECODE}(r)$;
$\text{UPDATECACHEDLOGITS}(s, \mathcal{Z}_{new})$;
return r;

Given an input request r, sampling configuration ϕ, and replay policy π, Algorithm 1 first obtains the intermediate state s and initializes $\mathcal{Z}_{new}$ for logits computed during this expansion. A cache hit provides two aligned sequences: cached next-token logits $\mathcal{Z}_{cache}$ and the tokens $\mathcal{Y}_{cache}$ generated along the cached continuation. A miss skips replay and proceeds to model inference. Keeping the continuation alongside its logits allows each replayed token to be checked against the path on which subsequent cached logits were computed.

For *Step-wise Sampling*, *Hive* traverses the cached logits in order and draws a token from every vector under ϕ. For *Hotspot Sampling*, it computes the score in Eq 4, min-max normalizes scores over the cached trace, and marks positions above the hotspot threshold; it draws a token only at those positions and copies $\mathcal{Y}_{cache}[i]$ elsewhere. Both policies append the selected token to r before comparing it with the corresponding cached token. On the first mismatch, replay stops with the divergent token retained in r: later cached logits condition on the original continuation and cannot be replayed for the diverged prefix. If the tokens match, replay continues to the next cached position.

After replay, a finished request returns without further model execution. Otherwise, *Hive* prefills the current r, whether the cache was missed or replay stopped or ran out, to materialize its KV state and produce the next logits. Normal autoregressive decoding then samples under ϕ until the request is finished. Each logit newly produced by prefill or decode is appended to $\mathcal{Z}_{new}$, which is used to update the cached logits associated with s; replayed logits are not regenerated. The procedure returns the completed request.

B Output Diversity of Hotspot Sampling

Hotspot Sampling aims to limit the loss of exploration while reusing stable portions of a previously generated continuation. The importance score in Eq 4 directs resampling toward positions with weakly peaked next-token distributions, while its time-decay term favors earlier positions, where a token change can influence a longer suffix. Answer correctness under this selective reuse is evaluated in Section 6; here we empirically evaluate how it affects output variation across sampling temperatures.

B.1 Token Overlap and Output Variation

To examine how the hotspot threshold affects output variation, we evaluate a single puzzle across ten temperatures $T \in \{0.1, 0.2, \ldots, 1.0\}$ and eleven thresholds $\tau \in \{0, 0.1, \ldots, 1\}$. At each temperature, two Hotspot Sampling branches share the same initial sequence and use distinct random seeds, each fixed across thresholds. We generate one output pair for each temperature–threshold combination, yielding 110 pairs in total.

For $0 < \tau < 1$, we resample positions whose normalized scores exceed τ; $\tau = 0$ recovers Standard Sampling, while $\tau = 1$ gives full reuse. Normal decoding resumes upon divergence.

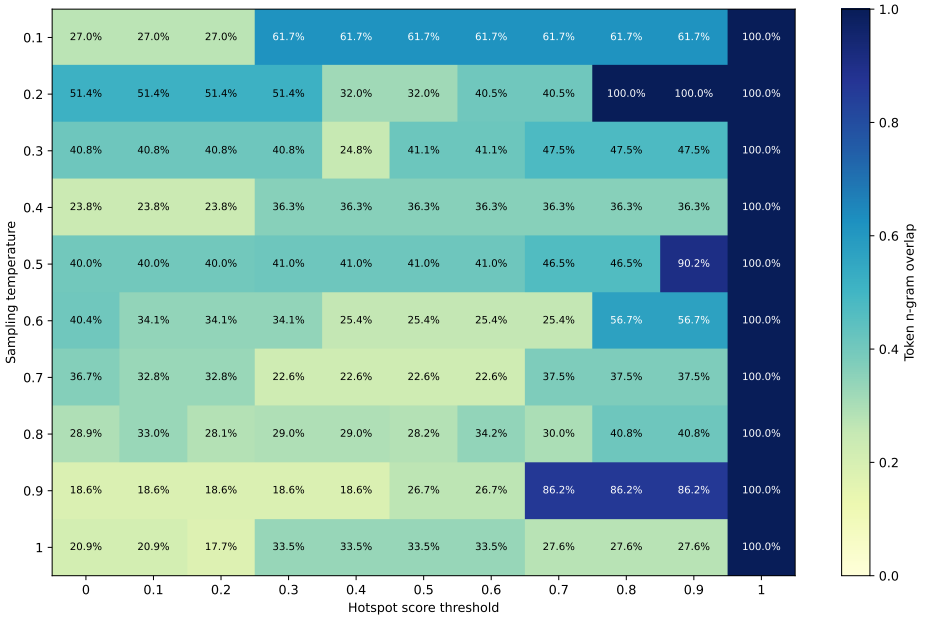


Figure 6: Token n-gram overlap between two Hotspot Sampling outputs across thresholds and temperatures.

After excluding prompt, padding, and termination tokens, we measure overlap between the two nonempty outputs u and v. Let $C_y(g)$ count occurrences of token n-gram g in sequence y, where $n = \min\{4, |u|, |v|\}$ and $|\cdot|$ denotes token sequence length. The normalized overlap is:

$$\text{Overlap}(u, v) = \frac{2 \sum_g \min\{C_u(g), C_v(g)\}}{\sum_g C_u(g) + \sum_g C_v(g)}. \tag{9}$$

Lower overlap indicates less shared local token structure. All output pairs are nonempty and retained.

Figure 6 shows that selective reuse retains token-level variation in the evaluated output pairs. At $\tau = 0.6$, mean overlap across temperatures is 36.3%, compared with 32.9% under Standard Sampling ($\tau = 0$), and every pair remains distinct. In contrast, $\tau = 1$ yields 100% overlap at every temperature. This comparison illustrates that selective resampling can retain output variation while reusing shared continuations.

C Front End

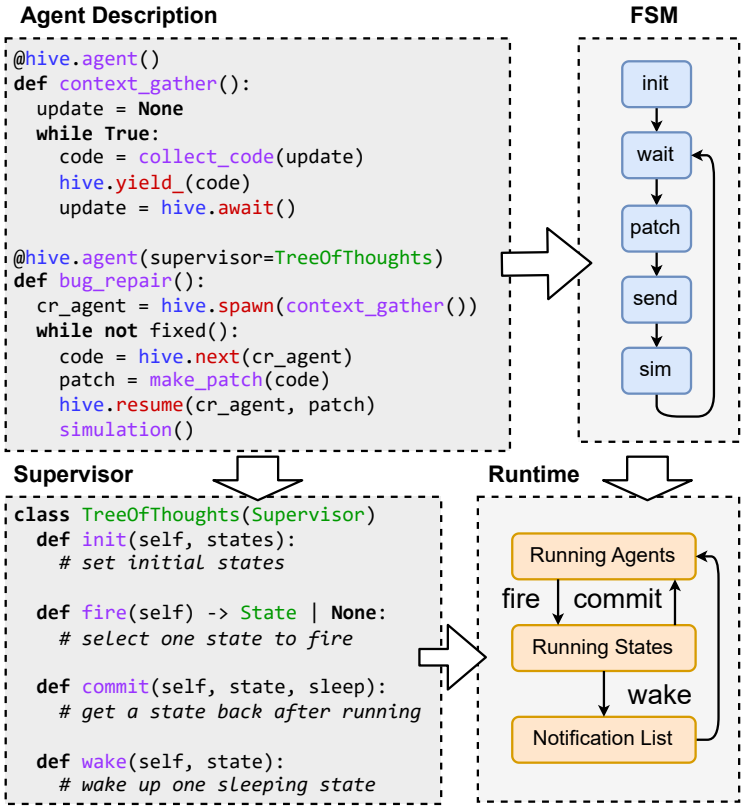


Figure 7: Hive Front End: the Python-based pseudo-code to describe agents and supervisors, the generated FSM, and the runtime system.

C.1 Agent Description and FSM

Hive describes the multi-agent system using an abstraction similar to coroutines in Python, to capture their asynchronous nature and simplify programming efforts. The workload of an agent can be described as a coroutine, which is a continuation of control flow while being suspendable to interact with other agents. An example of a program repair agent is shown in Figure 7. An agent is described as a function with the decorator **agent**. The *bug_repair* agent cooperates with the *context_gather* agent to help gather the related code snippets and fix the bug. The *bug_repair* agent first **spawns** the *context_gather* agent. Then, in each loop, it receives the gathered code from the *context_gather* agent by calling **next**, makes a patch by observing the code, and synchronizes the patch to the *context_gather* agent by calling **resume**. The loop continues until the patch passes verification methods like simulation. The *context_gather* agent is simpler. It keeps gathering code, **yields** the code to the *bug_repair* agent, and **awaits** the update.

Instead of directly executing the functions, *Hive* compiles them into FSMs. As shown in Figure 7, the FSM of the *bug_repair* agent exactly describes the behavior of the coroutine. In the FSM, a **state** represents the continuation. During one execution step, the agent transits to another state, inheriting the data and dialogue states for the LLMs. This process mirrors how Python interpreters compile async functions.

C.2 Supervisor and Asynchronous Runtime

Hive organizes test-time scaling algorithms in the form of **supervisors** to adapt to asynchronous execution. Each agent is equipped with one supervisor (as a parameter in the **agent** decorator). Instead of a single function, a test-time scaling algorithm needs to be implemented as an object with four callbacks. As shown in Figure 7, these callbacks are tightly integrated with the runtime system. When an agent is spawned, the **init** function provides the initial states of the FSM for the algorithm. Then, the runtime will keep querying the supervisor of a running agent whether to **fire** a state. If a state is fired, then the runtime will put it into the running states set and launch a real Python coroutine to transit the state in its underlying FSM. When the transition finishes, the new state will be **committed** to the supervisor. Certain mechanisms are implemented so the runtime can determine whether the transitioned state is sleeping, e.g., waiting for the result from another agent. If the transitioned state would yield data or finish the whole agent, the runtime will look up a notification list to inform the supervisors that some states should **wake** up, and can be fired if necessary. Such an asynchronous runtime allows multiple agents to run in parallel while keeping agent behavior orthogonal to test-time scaling algorithms, enhancing code reuse.